

DTIC FILE COPY

2

XINOTECH RESEARCH, INC.

TECHNOLOGY CENTER, SUITE 213
1313 FIFTH STREET SOUTHEAST
MINNEAPOLIS, MN 55414
(612) 379-3844

AD-A214 081

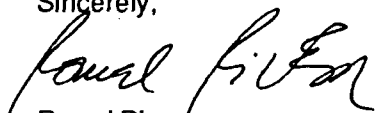
October 25, 1989

Defense Technical Information Center
Building 5, Cameron Station
Alexandria, Virginia 22314
Code S47031

Dear Steve:

We are enclosing a copy of the Progress Reports for the first two months of ONR's N89-001, Phase I. This is contract number N00014-89-C-0277.

Sincerely,



Rommel Rivera
Director of Research
RR/kh

DTIC
ELECTE
OCT 27 1989
S D CS D

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

ONR N89-001

A Prototyping Metalanguage with Formal Semantics for the Xinotech Program Composer

First Monthly Report
September 1, 1989

1. Progress Report

The following was the Performance Schedule proposed in the phase I proposal:

Task \ Month	0	1	2	3	4	5	6
Task I Objectives, XML	△	△					
Task II Design, XML	△	△	△	△	△		
Task III Objectives, XSSL	△	△	△				
Task IV Design, XSSL		△	△	△	△	△	△
Task V Prototype, XML				△	△	△	△
Final Report							△

N89-001 Phase I Schedule

Task I, Objectives of XML, is attached to this report.

Task II, "The Design of XML" has already been completed. This means that all major features of XML have been designed. Obviously, XML will undergo revisions during phase II and as we gain experience with it. The preliminary revision, however, is already complete.

89 10 26 110

Task V, "The XML Prototype" is 80% complete. This project was started at the time that the N89-001 phase I had been granted, since we did not want to run the risk of falling behind schedule. The only subtasks remaining are to complete the implementation of inheritance and write the "XML Reference Manual". Having the XML prototype available will allow us to also produce significant examples of its use, right during phase I. This will be done by creating a new task, Task VI, "XML Examples".

Task VI, "XML Examples" will show how languages may be defined using XML. Specifically, it will show CMS-2, Ada and ADADL. It will also show how inheritance may be used to provide support for embedding design languages into programming languages (Ada + ADADL) by reusing independent language components (the Ada language component and the ADADL language component). The Ada + ADADL definition will also show the use of views to isolate or combine the Ada and ADADL portions, as well as transformation views to show the ADADL-to-Ada migration of pseudo code. If possible, we would also like to show a language definition containing CMS-2 + ADADL.

Task VII, "Declaration Browser" is also a new task not included in the original phase I proposal. This will show how a simple prototype for incremental remote evaluation could be incorporated into XML in order to support interactive semantics to locate declarations locally or throughout libraries. Inter-file relationships are difficult to maintain using attribute grammars. Nevertheless, this is the single most important issue when dealing with semantics. At Xinotech, we wanted to show that we indeed view this issue as the most crucial one and that we see the feasibility of an efficient solution. The declaration browser was also started ahead of schedule and the language definition for it as well as its attribute evaluator have been implemented. What remains now is to write the attribute rules for Ada and CMS-2.

Our experience with Task VII has helped us substantially with the objectives and design for the semantical notation (Tasks III and IV). Both of these tasks are under way and we expect to have results according to schedule.

We would also like to add another new project, Task IX "XML



Accession For	
NTIS - GPOAL	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By <i>percs</i>	
DTIC/NSA	
Approved for Release	
Date	Authority
A-1	

Interface to Graphical Design". This task allows the user to define dependency relationships between compilation units and procedures using an XML notation, so that the Xinotech Environment can use such notation to provide a variety of flowgraphs using graphical design tools (e.g. CADRE, IDE). The XML flowgraph notation will be sketched, and the flowgraphs generated, except that in this prototype the XML notation will not be used to generate the flowgraphs. Flowgraph generation will be "hardcoded" into the Xinotech Environment. This is a task which will be shared with NSWC's N89-149. In the NSWC phase I, it is justified as a mechanism for reverse engineering the high level (graphical) design. Here in the ONR's N89-001 phase I research, it is justified as an example of how the flowgraph generation can be specified in a language independent fashion using XML.

2. Refinement of Phase II Goals.

During our August 23 meeting, Xinotech stated that the goals for XML during phase II were to support:

- * embedded languages
- * language prototyping
- * language reusability
- * interactive semantics
- * language customization
 - * multiple views, view sets
 - * transformation views
 - * cascading (abstract level migration)
 - * protecting language integrity
- * environment generation (menus, p-holders)
- * integration
 - * compilers, graphical, design, pdls, debuggers
- * language version control
 - * upgrade abstract trees
 - * compatibility across hosts and versions

Upon reviewing the scope of such projects and their relevance in an integrated software environment, we feel that all these goals for XML are viable and realizable during phase II of this research. This will be justified later.

ONR N89-001 Phase I

**Task I. Definition of the Requirements and Objectives for the
Design of the Metalanguage XML.**

September 1, 1989

This paper contains the goals and requirements for the design of XML. This excludes the design of the semantic notation which is addressed in tasks III and IV.

The metalanguage is the one, comprehensive formalism needed to instantiate all the language-based aspects of the environment. The problem of language complexity will be managed by organizing XML so as to allow both the language designer and end user to express the major aspects of the language in the simplest possible terms, while providing powerful constructs that may be used by the sophisticated designer in order to support the rigorous demands of a practical environment. This approach to the design of XML will be conducive to language prototyping.

XML will be designed to provide support in the following areas:

1. Language Design.

1.1. Readability.

Language descriptions will be easy to read and easy to document. The XML definition of a language construct will show the abstract grammar, external syntax, and unparsing (and other) views in a concise form that resembles the "template" or actual appearance of the construct in a program. Optional components will be clearly and simply marked. Each detail of formatting for an unparsing view will be marked where it occurs in the syntax, using a small but powerful set of simple commands. Comments will be allowed anywhere in the language definition. To make it readable, XML will be given the syntactic flavor of languages such as Ada and Pascal. An example appears on the next page.

```

construct ifStmnt
  components
    { This is the abstract syntax for an IF statement. }
    condition      : expression;
    ifPart         : stmtSequence;
    elsifParts     : [elsifPartList];
    elsePart       : [stmtSequence]
  view ada (template) scheme
    { External syntax --- doubles as main unparsing view }
    ( _if _ ?> condition )
    ' ' ?| _then_ > ifPart < [ elsifParts | ]
    [ _else_ > elsePart < ] _end _if_ ' ';
  view summary scheme
    _if _ condition _ then_ { Summarized unparsing view. } .
  view "list-summary" scheme
    _if_.
  placeholders
    construct
      if_stmnt
    components
      condition, statements, elsif_parts, statements
end ifStmnt

```

Example: Ada if-statement

Curly brackets {} contain comments. Square brackets [] show optional components. Short commands, such as "|" (line fold) and "?>" (optional line fold with indent), control formatting. In the view schemes, square brackets clearly show which formatting commands are to be invoked (and which keywords are to appear) when an optional component is present. The designer controls the placeholders that will appear in templates.

1.2. Prototyping Languages from Other Notations.

Languages will be easy to prototype from other notations. The abstract syntax will be closely related to the extended Backus-Naur form (EBNF), which is the model for syntax description in most meta-languages. The XML transformation views facilities will be available to the user to semi-automate the process of producing an XML notation from other analogous notations such as EBNF.

1.3. Prototyping Languages from Previously Defined Languages (Language Reusability).

Languages will be easy to prototype from previously defined languages. XML will support: (i) division of a language into re-usable modules; (ii) module import and inheritance; (iii) construct inheritance. These will make it easy to re-use parts of an existing language, with or without modifications. Module inheritance can be used for coarse-grained modifications, and construct inheritance for fine-grained changes.

When Ada is to have embedded customized PDL's, it is important that each language can evolve separately from the other. Ada may be supported by the vendor (e.g., Xinotech), whereas the PDL may be supported internally by the customer. If the metalanguage does not support the reuse of separate components, both languages would have to be supported by one party, either the vendor or the customer, a highly undesirable proposition. If a language needs to be supported by the customer (e.g., in the case of a confidential PDL), Ada ends up being supported twice.

1.4. XML, A Powerful Notation.

XML will be powerful enough for the advanced language designer. By permitting arbitrarily many views, XML will allow the designer to tailor the appearance of programs to the needs of many users and installations in one language description. Views can also be used to produce transformed versions of programs that resemble other languages, or that support programming tools, such as compilers. For example, it will be possible to use XML to define the granularity for incremental compilation. XML will also provide view commands for sophisticated control of formatting, such as hierarchical line folding that varies according to the distance to the right margin, as in the following example. Consider the Pascal for-loop statement:

```
FOR abc := base TO lim BY incr1 + incr2 * incr3 DO x := y END
```

As the horizontal space diminishes, we might want this statement to fold first to this (next page):

```
FOR abc := base TO lim BY incr1 + incr2 * incr3 DO
```

```
  x := y
```

```
END
```

then to this:

```
FOR
```

```
  abc := base TO lim BY incr1 + incr2 * incr3
```

```
DO
```

```
  x := y
```

```
END
```

next to this:

```
FOR
```

```
  abc := base
```

```
  TO lim BY incr1 + incr2 * incr3
```

```
DO
```

```
  x := y
```

```
END
```

then to this:

```
FOR
```

```
  abc := base
```

```
  TO lim
```

```
  BY incr1 + incr2 * incr3
```

```
DO
```

```
  x := y
```

```
END
```

and, finally, to this:

```
FOR
```

```
  abc := base
```

```
  TO lim
```

```
  BY
```

```
    incr1 + incr2 * incr3
```

```
DO
```

```
  x := y
```

```
END
```

XML will permit view schemes to express such subtle control, using only a small repertoire of margin control and folding commands.

1.5 Language Decomposition and Embedding.

XML will be useful for decomposing and embedding languages. XML's modules will encourage the splitting of a language description into re-usable portions. In addition, XML will allow nesting of language constructs, so that, for example, the definitions of all the parts of a procedure declaration could be nested inside the construct that defines "procedure declaration". This encourages top-down, modular design, which makes the language easy to read and easy to maintain. The import and inheritance mechanisms will make it simple to embed a design language into another language. Conversely, these mechanisms will make it possible to strip an embedded design language from another language.

2. Language Semantics.

This topic will be covered in tasks III and IV of phase I. However, a prototype attribute notation is being incorporated into XML, in order to allow us to provide inter-file navigation through object declarations. With this notation, XML will provide a mechanism for specifying where declarations occur in a program, and whether their scope is local or global. This will allow the Composer to browse for objects with a given name. In this way, the user in the midst of constructing a program can find all the declarations of an identifier without having to search through existing program files.

3. Language Specification and Environment Generation.

A single XML language definition will describe the syntax of the language, the semantics of declarations, menus for selection of alternative constructs during editing, and placeholders for templates. Furthermore, for each construct, all these specifications will appear together, in one readable declaration.

4. Language Customization.

4.1. By using import and inheritance, the designer will be able to produce a language that differs from an existing language in some aspect of the abstract syntax, without extensive rewriting.

4.2. XML will make it easy to add views to an existing language. Specialized views that affect only a few language constructs will not require extensive revisions: instead, the designer will need to write view schemes using the new views only for those language constructs where the customization is relevant; other constructs will be covered through a flexible default mechanism.

4.3. XML will permit formatting views that produce localized effects. For example, a formatting view "vertical_params" might be written to align procedure and function parameters in columns. By specifying such formatting views in the declaration of a global view, the designer will be able easily to produce a view that reflects a particular mix of formatting preferences. Furthermore, through a "view alias" mechanism, XML will permit the specification of variant views that differ from existing views only in the choice of formatting views, or in certain global formatting details such as indentation size and line length.

5. Program Transformation.

5.1. Batch Language Conversion.

XML will support language conversion, through the use of unparsing views that represent a program in a different external syntax. This capability of XML will be of great advantage in NSWC's N89-149 Reverse Engineering research since it will facilitate conversion to other languages as well as the targetting of the extracted design to specific design languages.

5.2. Interactive Construct Transformations.

By supplying a command to insert a placeholder, for use in view schemes, XML will make it possible to write transformation views that will automatically transform from one construct to another.

For example, this facility will allow the transformation of an Ada package specification into a package body, or a procedure header into a procedure body, or the automatic production of a procedure call (with association parameters) from the procedure's declaration.

This facility will be unique in that:

a) It will be customizable by the user. Transformations are specified as part of the language definition. Therefore the user will be able to specify other transformations of interest, for example:

- i. "case" to "if" statements or "if" to "case" statements;
- ii. Certain forms of minimization of boolean conditions.

b) It will language independent. Transformations of the kinds suggested here can be specified for any language.

Our solution to program transformation is not a "hardwired" solution to a special case or a specific language. This is a general mechanism exploiting our essential concepts of language independence and multiple language representations.

5.3. Cascading or Migrating Through Levels of Abstraction

One particular application of 5.2 is the potential to generate, from the code at a given level, the initial skeleton for the corresponding code at the next level of abstraction. For example, a high level design for an Ada program could be used to generate the first approximations to the specification packages, and they in turn, once supplied with detailed level design, used to generate package body skeletons. They in turn, once complemented with the algorithms expressed in an embedded design language, could be used to generate the initial Ada code skeletons. In this way, the Composer supports "cascading" through levels of abstraction.

6. Integrity of Language Versions.

When import and inheritance are used to produce a customized language, or to embed one language in another, only a single copy of each previously defined language need be maintained. This has several benefits: no copying of languages or modules is required; upgrades to the existing languages are automatically carried over to the new ones; because the integrity of the existing languages is not violated, responsibility for maintenance of languages can be cleanly divided.

7. Line Formats.

Some languages, for instance COBOL, require that each line of a program be formatted in a certain way. For example, certain columns may be reserved for some purpose. XML will provide an optional mechanism for the detailed control of column use. Other languages, such as Ada, may use the end of a line to delimit a comment. XML will provide view commands to control the recognition of comments that belong to only one line, and to insert appropriate strings when long comments are broken between lines. Although such line-oriented issues as comments in Ada seem simple, they are actually difficult for language-oriented tools to handle, and it requires considerable thought to arrive at elegant solutions.

8. Environment Integration.

8.1. Incremental compilation.

Assume that an incremental compiler, say for Ada, has been written, and that it accepts certain minimal, indivisible program fragments ("increments") for recompilation, for example procedure or function bodies. If the XML description of Ada specifies what language constructs are acceptable increments, the Composer can keep track of the increments while the program is being edited, and submit to the compiler only those that need to be recompiled.

8.2. Protection of the integrity of graphical design.

Graphical Design Tools are used to specify high level design, package decomposition and interfaces. Some of these tools support the transition from design to text, by providing the

textual skeletons equivalent to the graphical design. By permitting the language designer to specify what constructs are part of the skeletons, XML can make it possible to protect a skeleton against editing changes that violate the design. Conversely, if editing is to be allowed to change the graphical design, XML will permit the language designer to write views that generate skeletons (sometimes called "picture scripts") acceptable to the graphical design tool.

9. Language Version Control.

9.1. Language Evolvability.

The compiler for XML will provide a tool that automatically upgrades existing program files belonging to a language when the XML description of that language changes. When languages are rapidly changing, as may be the case with newly designed, customized or proprietary PDL's, it is important that the language's evolution does not make obsolete the software already written. There have to be mechanisms automatically and transparently to upgrade the software to the new versions of the language. Without this feature, companies will be reluctant to use language-based composition tools.

9.2. Language Time-Stamping.

When descriptions of the same language are prepared on different hosts, the question arises whether program files prepared using the language generated from the description on one host may be transferred to the other host(s). By providing for a unified version control mechanism, XML can make it possible to identify which program files are truly compatible across hosts.

**A Prototyping Metalanguage with Formal Semantics for the Xinotech
Program Composer**

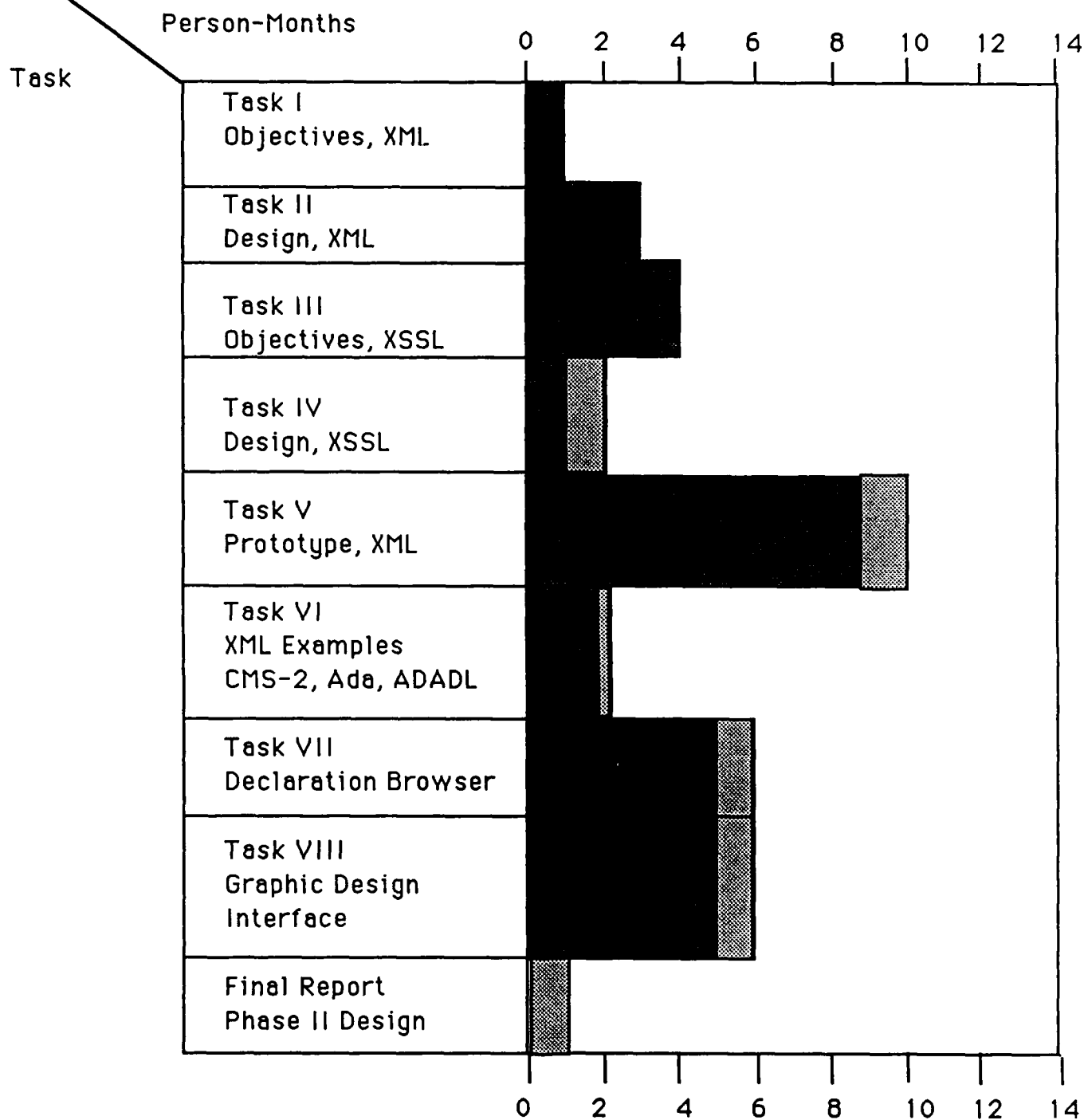
Second Monthly Report
October 1, 1989

1. Progress Report

As specified in the Phase I proposal schedule, we are delivering Task III, "Objectives, XSSL."

Task VIII, the Graphical Design Interface prototype is functional now. It generates graphs for both Ada and CMS-2, using IDE's Software Through Pictures (we have been unable to obtain a copy of CADRE's TeamWork). We are now in the process of specifying the full phase II design for this tool and writing the documentation.

The following chart shows the comparative progress up to this point:



**ONR's N89-001 Phase I
Month 2 Progress Chart**

ONR N89-001 Phase I
A Prototyping Metalanguage with Formal Semantics for the Xinotech Program
Composer

Task III
Definition of the Requirements and Objectives for the Design of the
Semantic Notation XSSL.

October 1, 1989

Abstract

This paper contains the goals and requirements for the design of XSSL, the semantic notation for XML.

The Xinotech Semantic Specification Language (XSSL) will be the portion of XML (the Xinotech Meta-Language) devoted to specifying the semantic properties of languages needed to instantiate language-based programming environments. This programming environment in question is the Xinotech Program Composer.

The Objectives for XSSL are:

Providing a high-level notation for formal specifications of semantics, Providing for a time and space efficient implementation of the notation, Providing a notation that is well suited for the construction of interactive language-based programming environments.

Contents

1. Introduction

2. Objectives

2.1 Reducing the Complexity of the Formal Specification

2.2 Focus on Interactive Semantics

2.2.1 The Specialized Notation Should be Targeted for Interactive Use

2.2.2 Reducing Execution Time and Memory Utilization

2.2.3 Specific User Interface

3. Requirements

3.1 Language-Specific Semantics

3.1.2 Use-Declaration Facilities

3.1.3 Scope and Visibility Rules

3.1.4 Generality

3.2 Clarity

3.2.1 Integration with XML

3.2.2 Scalable to Large Specifications

3.2.3 Declarative Language

3.3 Interactive Semantics

3.3.1 Program Display and Manipulation

3.3.2 Program Navigation

3.4 Timing and Capacity

3.4.1 Response Time

3.4.2 Incremental Evaluation

3.4.3 Efficient Changes to Aggregates

3.4.4 Large Collections of Program Units

3.5 Language Prototyping

3.5.1 Easy Modification of Specifications

1. Introduction

The Xinotech Semantic Specification Language XSSL will be the portion of XML devoted to specifying the semantic properties of languages. Although XSSL is an integral part of XML, its design is considered in a separate task because semantic formalisms are a different technology than syntactic notations.

The need for XSSL comes from the need to use semantic information in a language based software environment. A language-based editor that uses syntactic information allows a user to manipulate a program structurally. Such an editor can make sure that a program is structurally correct - that is, that each construct is correct according to the syntax rules, given knowledge of the construct's immediate surroundings in the program. But, a language-based programming environment needs to make use of the non-local context of a construct. For instance, an environment might allow manipulations or enforce language rules using type information about an object, or knowledge of the kind of program unit that contains a particular construct. That information can't be found locally, but must be brought from a remote location in the program - the declaration section, or the procedure header. Such non-local information is called the static semantics of a program. Since XML is a meta-language for describing language based software environments, it needs to describe the static semantics of languages.

XSSL can also be used to support the implementation of interactive environment commands which rely on the resolution of the static semantics of the language. Examples include: 1) a facility for navigation through declaration dependency chains, or 2) a command to complement a record field reference with an automatic expansion of its preceding designator sequence (qualifiers, array references, dereferences).

2. Objectives

Attribute grammars have been the formalism studied the most for expressing static semantics. However, they have not reached the maturity necessary to be applied in practical software environments. These are the reasons:

a) Their Inherent Complexity. Specifying the semantics for a language using attribute grammars is a comparable task to writing a compiler for that language. Even though attribute grammars provide the advantage of producing a formal specification of the semantics of a language, it comes at no savings in the complexity of the resulting product. In addition, the unstructured, production-oriented form of traditional attribute grammar notations make them unsuited for creating large, reliable language descriptions.

b) Their Exorbitant Consumption of Computational Resources. The CPU and memory requirements for processing attribute grammars is very high.

The use of inherited and synthesized attributes to specify flow through the nodes of a tree requires that every node in the tree be a participant in this flow, even if only as intermediaries for passing attributes otherwise irrelevant to them. This is the inherent reason why attribute grammars impose such an exorbitant space requirement.

This could be best exemplified by the fact that, to the best of our knowledge, a software environment with formal semantics for a language with the complexity of Ada has not even been attempted.

In stating our objectives we need to find solutions or alternatives to these two drawbacks.

2.1 Reducing the Complexity of the Formal Specification

The typical semantic issues of programming languages will be characterized so that a specialized notation, SN, for such issues can be developed. Its relationship to the general formalism selected, GF, will also be shown. The specialized notation SN is to be designed so that it can coexist with the general formalism GF, as part of the XSSL. They both should integrate into the single XSSL. This arrangement presents the following advantages:

- a) The size and complexity of the description for the most needed semantic features of a language will be reduced by an order of magnitude.
- b) When the specialized notation SN is used, the expressing power is amplified, enhancing readability.
- c) It allows easy prototyping of the semantic features of the language that are most frequently used interactively.
- d) The general mechanism is still available whenever it is appropriate to complete the semantic details of the language.
- e) The specialized notation SN can be mapped to the more general formalism from which it was derived by the automatic means of a translator whenever this step may be required, for example, to make use of general evaluation algorithms, or to give uniform treatment internally to the SN and GF notations.

2.2 Focus on Interactive Semantics

2.2.1 The Specialized Notation Should be Targeted for Interactive Use.

The first priority, when designing the formal specification of semantics, is that it be used effectively to help automate the process of the interactive construction of programs. In this situation, what is needed most is the ability to describe the scoping rules of the language so that interactive name resolution may be done. This would facilitate the construction process by allowing automatic navigation through the complex inter-relations of software systems and libraries to locate objects that need to be looked up or referenced during program construction.

We feel at Xinotech that it is our responsibility to integrate with existing compilers and use interactive semantics not to replace the functionality already provided by compilers, but rather, to support the role of the front end composition tool (the Composer) in the whole environment. The result is emphasis on techniques to support program construction and manipulation, and less emphasis on error correction (type checking) and code generation. Some examples of these support techniques are program navigation and interactive transformations. Program navigation techniques are used to locate declarations locally and throughout libraries to provide knowledge about the system to the programmer. Interactive transformations provide language specific commands, for instance to create a procedure call template from a procedure declaration.

2.2 Reducing Execution Time and Memory Utilization.

Application of static semantics during interactive program construction has a higher priority than other potential uses such as formally specifying code generation (dynamic semantics). This is an important consideration given that the use of formally specified program synthesizers to replace carefully targeted code generators for complex languages such as Ada and CMS-2 cannot even be considered in the short term.

For such a case, incremental semantic evaluation on a partially attributed tree may be ideal in combination with the specialized scoping mechanism to explicitly retrieve, for example, the specific object declarations needed for inspection or expansion during program construction. The use of a specialized notation with no explicit attribute flow would substantially reduce the space and time requirements.

2.2.2 Specific User Interface.

This interactive semantic scheme will make it possible to design a user interface that provides access to the semantic information within the abstract trees.

3. Requirements

Scope

These are the requirements for XSSL. Also, some requirements are listed for the evaluator that will be used to evaluate XSSL. This is done since the design of XSSL and the evaluator influence each other, and so some requirements for the evaluator impose requirements for XSSL.

3.1 Language-Specific Semantics

XSSL has to be able to specify the language-specific semantics of a software development environment.

3.1.2 Use-Declaration Facilities

XSSL must be capable of specifying "use-declaration" facilities. A "use-declaration" facility is a facility that can provide the declaration(s) of an identifier that are visible at a particular point in a program. The ability to specify use-declaration facilities has been chosen as the immediate goal of this project. This is so because the ability to find the declaration of an object in a program that consists of a large number of modules, including library units, is an important tool for the development and maintenance of large programs.

It is necessary to be able to find the declarations of an object that are visible at a use, to help in understanding an existing use of an object.

It is also useful to be able to find all the declarations of a particular identifier. This is helpful when changing a program, to look at the declarations that are available but not yet visible at a given point.

3.1.3 Scope and Visibility Rules

XSSL must be capable of specifying language scope and visibility rules. Because of requirement 3.1.2, an environment supporting a language specified in XSSL needs to make use of the scope and visibility rules of the language. This will be restricted to languages with static scope and visibility; since a language specific environment needs to manipulate a program before it is executed, dynamic scope is outside the range of this proposal.

3.1.4 Generality

XSSL must be general.

Although the initial goal of XSSL is to specify use-declaration facilities, it should not be a tool for creating language-dependent facilities to compute use-declaration information. Instead, it should be a more general notation in which use-declaration facilities can be expressed. There are two reasons for this requirement. The first is that a specialized tool that can be parameterized to several different languages runs the risk of not being general enough to handle new languages. The second is that this project should be the base for other more advanced applications, and so a more general notation is needed.

It may be the case that there are conflicts between the need for generality and other requirements, in particular the requirements for conciseness and efficient implementability. For that reason we've introduced the idea of extending a general notation (GF) with a specialized notation (SN) where necessary.

Our primary goal is the support of program construction and maintenance. To meet that goal it is important to efficiently support features that help understand large programs, including:

- o find the declarations visible at a use
- o find all declarations of a name
- o find all references to an object for a declaration of a name,
- o find the implementation (for instance, from a declaration in an Ada package specification, find a specification in the package body)

Where needed the specialized notation will have features to efficiently support those features.

There are many potential applications for interactive semantics in a programming environment. These include:

- o translation from one source language to another (For instance, translating from a Modula-2 WITH statement to Ada requires static semantics)
- o interactive transformations of constructs (case statement to if statement, procedure declaration to procedure call template, etc.)
- o automatic type checking
- o checking of other static semantic constraints (actual parameters matching formal parameters, etc.)
- o user-defined pretty printing of programs
- o generation of interpretive code
- o data flow analysis
- o production of input to analysis tools such as graphical design tools

The general notation is intended to be able to support these.

3.2 Clarity

XSSL specifications must be clear and simple

3.2.1 Integration with XML

This is necessary to manage the complexity of a large semantic specification. XML's modularity allows a language definition be decomposed into understandable parts. Since XSSL specifications will be an integral part of XML definitions, they can also be written as modules. Also, the integration of XML and XSSL will allow XSSL specifications to be part of hierarchy language definitions, for example, an Ada specification embedded into an Ada+PDL specification.

3.2.2 Scalable to Large Specifications

XSSL must not be overwhelmed by large semantic specifications. The size of a specification should be proportional to the complexity of the semantics being described. So, the notation should not include book-keeping information that grows more than linearly with an increase in the size of the specification. An example of that is the use of "copy-rules" in a traditional attribute grammar.

3.2.3 Declarative Language.

Semantic specifications for a language-specific environment should be declarative, not procedural. This requirement is needed to get incrementality (See section 3.4). If the semantics is defined procedurally, then evaluating the semantics for a part of a program has an effect that is defined by a procedure. In general, procedures can interact in ways that depend on the order in which they are executed. If a part of the program is deleted, the effect of the corresponding procedure has to be "undone" somehow. That is difficult, since the effect of a procedure depends on the other procedures that are executed before and after it. If the definition is declarative, however, then the effect of each rule of the definition is not order dependent, it is only dependent on the explicit inputs to the rule. So the evaluator can "undo" each rule in a controlled way.

If a procedural specification is used, explicit inverse procedures can be provided to undo the effect of each procedure, but that is difficult and error prone.

3.3 Interactive Semantics

XSSL must be focused on the task of specifying interactive semantics. That is because the purpose of semantic specifications in XML is to help automate the process of the interactive construction of programs.

3.3.1 Program Display and Manipulation.

The purpose of specifying semantics in a programming environment is to support the interactive construction of programs. So, there has to be a way of using the information defined by a semantic specification to provide interactive tools. So, the language-specific programming environment needs access to the values specified in XSSL.

The interfaces needed to XSSL values include the following:

There must be a way to display specific points in a program. For instance, an XSSL specification may describe the declarations corresponding to a use of an object. The environment has to be able to display the part of the program containing those declarations.

There must be a way of displaying messages computed by an XSSL specification, and relating messages to points in a program. For instance, if an XSSL specification computes violations of type rules, the environment must display an error message when a violation is found, and must display it with the point in the program where the error occurred.

The XSSL computed values must be usable to control the `_view_` used to display the program. For instance, an XSSL specification could be written to specify an indentation scheme, and the values computed would control the display of the program. Another use for XSSL values is in translation views, which are used to translate from one source language to another. Semantic information needed for such a translation would be computed using XSSL and used in the result of the translation.

XSSL values must be usable to control interactive transformations. Interactive transformations convert a language construct to a related one. For instance, a transformation might be defined to convert a procedure declaration into a template for a procedure call. When such a translation requires semantic information, values computed with XSSL need to be used in the transformation.

3.3.2 Program Navigation

XSSL must support movement inside and between programs. Semantic values defined in XSSL must support a facility for navigating from a use of an object to its definition (which may be in a different file or library unit from the use.) So there must be a way of defining interfaces between XSSL and the environment, so that the environment can use the values defined in XSSL to provide a navigation facility.

In a system intended for large-scale, practical software engineering use, it will be necessary to move from one program unit to another on the basis of semantic information. For instance, given a use of an object, the environment will need to display the declaration of the object, even if it is declared in a library, or another program unit. In some applications the set of program files to be considered will be very large. The problem of keeping track of the connections between a large set of files and navigating among them has not received much attention in the past. Work on attribute grammars has usually assumed that attributes are computed over the abstract syntax tree for a single program.

3.4 Timing and Capacity

The evaluator for XSSL must work interactively, with large software systems.

3.4.1 Response Time

There must be an evaluator for XSSL that provides reasonable response time.

When a user of an environment makes a change to a program, the values defined by the semantic specification must be re-evaluated quickly enough to provide a reasonable response time.

3.4.2 Incremental Evaluation.

The complete re-evaluation of an XSSL specification after each change to a program will take too long. So, the evaluator for XSSL must work incrementally. When a change is made to a program, an incremental evaluator saves time by re-using the values that are not affected by the change. The point of incremental evaluation is to update the semantic information for a program after each change in less time than re-evaluation of the specification for the entire program.

3.4.3 Efficient Changes to Aggregates

The evaluator for XSSL must evaluate incremental changes to aggregates efficiently.

An "aggregate" is a single object that represents many smaller pieces of information. An example is a symbol table, which represents many declarations. Re-evaluation after a small change to an aggregate should be fast, and so should not take time proportional to the size of the entire aggregate. For instance, if a declaration in a symbol table is changed, the time to re-evaluate the semantics should be small and close to constant, not close to the time needed to re-create the entire symbol table. This requirement is a result of the requirement for incremental evaluation.

3.4.4 Large Collections of Program Units

The evaluator for XSSL must be able to handle large programs, consisting of many files and library packages.

It has to be able to do this interactively, so a small change to a library cannot require immediate re-evaluation of semantic information for all the modules that use the library. So, the evaluation method needs to be able to reduce or defer the re-computation required by a change that has a global effect.

3.5 Language Prototyping

The evaluator for XSSL should support language prototyping.

3.5.1 Easy Modification of Specifications

The evaluator for XSSL should be able to respond quickly to a change in an XSSL specification.

The semantic specification for a language should be separate from the code implementing the attribute evaluator and the environment. Changing a specification should not require re-compilation of the environment or evaluator. There are two reasons for this requirement :

A single environment should be able to handle multiple languages, for the convenience of the user, and

Re-compilation of the entire system will make the job of tailoring the environment to a new language difficult, because a small change made while debugging a language specification will require a time consuming recompilation.